Junior .NET Developer — Complete Interview Guide
• OOP • SOLID • C#

36 essential interview questions covering OOP, SOLID Principles, C# Type System, Exception Handling, and more — designed for Junior .NET Developer roles.

Section 1: Core OOP Concepts

Q1: What is the difference between Abstract class and Interface?

	Abstract Class	Interface
Methods	Implemented + abstract methods	All abstract (C# 8+ allows default)
Fields	Can have fields	No fields
Inheritance	Single inheritance only	Multiple interfaces allowed
Constructor	Has constructor	No constructor
Use when	Shared base behavior	Define a contract

Q2: What is the difference between Overload and Override?

Overloading → Same method name, different parameters, same class.
Overriding → Child class redefines a virtual/abstract method from parent.

Q3: What are OOP Principles?

4 Pillars:

- Encapsulation – Hide data, expose via properties
- Inheritance – Child class reuses parent behavior
- Polymorphism – Same method, different behavior
- Abstraction – Hide implementation, show only what's needed

Q4: What are SOLID Principles and how do they relate to OOP?

Letter	Principle
S	Single Responsibility (one job only)
O	Open / Closed (open for extension, closed for modification)
L	Liskov Substitution (A child class must work wherever the parent)
I	Interface Segregation (Split big interfaces into small focused ones)
D	Dependency Inversion (High-level & Low-level classes depend on an interface or abstraction)

S — Single Responsibility: Each class has one job only. OrderService → only handles orders. Not emails, not reports.

O — Open/Closed: A class is open for extension, closed for modification. Add new behavior by creating a new class, not editing the old one.

L — Liskov Substitution: A child class must work wherever the parent is used — without breaking anything. If Dog inherits Animal, using Dog instead of Animal should still work fine.

I — Interface Segregation: Don't force a class to implement methods it doesn't need. Split big interfaces into small focused ones.

D — Dependency Inversion: High-level classes should not depend on low-level classes. Both depend on an interface/abstraction, not a concrete class. → This is exactly what Dependency Injection is based on.

SOLID makes OOP code maintainable, scalable, and testable.

Q5: What is Encapsulation and how do you implement it?

Encapsulation = protecting data by making fields private and accessing them through public properties.

Q6: What is a Property and how does it achieve Encapsulation?

A property is a controlled access point to a private field. It enforces rules on get/set — e.g., validation before assignment.

Q7: What is a Constructor and what is its benefit?

A constructor is a special method that runs automatically when an object is created. Same name as the class, no return type.
Benefits:

- Initialize object state
- Enforce required values at creation

Q8: What is the difference between Abstract, Sealed, and Static class?

	Abstract	Sealed	Static
Can instantiate?	No	Yes	No
Can inherit?	Yes (must)	No	No
Purpose	Base class with incomplete logic	Prevent inheritance	Utility class, no instance needed
Has normal methods?	Yes	Yes	No

Has abstract methods?	Yes	No	No
Has static methods?	Yes	Yes	Yes (only)

Q9: What is a sealed method? How is it different from a sealed class?

sealed class → prevents the entire class from being inherited.
sealed method → prevents a specific overridden method from being overridden again in further subclasses. Must be used with override.

⇨ *Use sealed methods when you want to allow inheritance of the class but lock down a specific behavior.*

Q10: What is the difference between Abstraction and Encapsulation?

	Abstraction	Encapsulation
Definition	Hiding complexity — show only what is needed	Protecting data — control access to it
Focus	Design level — what the object does	Implementation level — how data is stored
Achieved by	Abstract classes, Interfaces	Private fields + public properties
Example	Drive() hides engine details from the driver	Speed is private, exposed only via GetSpeed()

Q11: What is a Static method and Static variable?

- Static variable → Shared across all instances (one copy in memory)
- Static method → Called on the class itself, not on an object. Can only access static members.

Q12: What is the benefit of OOP?

- Reusability → Inheritance
- Security → Encapsulation
- Flexibility → Polymorphism
- Maintainability → Abstraction + SOLID
- Scalability → Modular structure

Section 2: Polymorphism & Inheritance

Q13: What is Polymorphism? (Runtime vs Compile-time)

	Compile-time	Runtime
Also called	Static Polymorphism	Dynamic Polymorphism
Achieved by	Overloading	Overriding

Decided at	Compile time	Runtime

Q14: What is the virtual keyword?

Allows a method in the parent to be overridden in the child.

⇨ *Without virtual → child cannot override it.*

Q15: What is the difference between override and new in inheritance?

	override	new
Replaces parent method?	Yes	No — Hides it
Polymorphism works?	Yes	No

Q16: What is this vs base keyword?

- this → refers to the current object
- base → refers to the parent class

Q17: What is a Destructor?

Called automatically when object is destroyed / garbage collected.

```
~Dog() {
    Console.WriteLine("Dog destroyed");
}
```

⇨ *In C# we prefer IDisposable + using statement over destructors.*

Section 3: Classes & Modifiers

Q18: What are Access Modifiers?

Modifier	Accessible From
public	Anywhere
private	Same class only
protected	Same class + child classes
internal	Same project only
protected internal	Same project + child classes

Q19: What are Constructor Types?

Type	Description
Default	No parameters, auto-created if none defined
Parameterized	Takes parameters to initialize object
Copy	Creates object from another object
Static	Runs once, initializes static members

```
public Person() { }                    // Default
public Person(string name) { }         // Parameterized
public Person(Person other) { }        // Copy
static Person() { }                    // Static
```

"Compiler automatically provides a default constructor if no constructor is defined. But if any constructor is explicitly defined, default constructor is not provided."

Q20: When to use Abstract class vs Interface?

Use Abstract Class When	Use Interface When
Classes share common code	Just defining a contract
You need a base implementation	Multiple inheritance needed
Classes are closely related (same type)	Classes are unrelated (same behavior)

Rule of thumb:

- Abstract class = "is a" relationship → Dog is an Animal
- Interface = "can do" relationship → Dog can ISwim

Q21: What are Inheritance types in C#?

- Single Inheritance → class Dog : Animal
- Multi-level → Dog : Animal : LivingThing
- Multiple (interfaces) → class Dog : Animal, ISwim, IRun

Multiple class inheritance is NOT supported in C#.

```
// Allowed
class Dog : Animal, ISwim { }
// Not allowed — Error!
class Dog : Animal, Pet { }
```

Q22: If a class inherits another — which constructor runs first?

The Parent (Base) class constructor always runs first.

```
// Output: =>  Animal Constructor =>  Dog Constructor
```

Section 4: SOLID Principles (Detailed)

Q23: Explain the Single Responsibility Principle (SRP)

A class should have only ONE reason to change — only one responsibility.

Q24: Explain the Open/Closed Principle (OCP)

Open for extension, Closed for modification. Add new behavior by adding new classes, not by editing existing code.

Q25: Explain the Liskov Substitution Principle (LSP)

A child class must be usable wherever the parent is used — without breaking the program.

- ⇨ *Rule of thumb: if the child throws NotImplementedException or overrides with empty body → LSP is violated.*

Q26: Explain the Interface Segregation Principle (ISP)

Don't force a class to implement methods it doesn't need. Split large interfaces into smaller focused ones.

Q27: Explain the Dependency Inversion Principle (DIP)

High-level classes should not depend on low-level classes. Both should depend on abstractions (interfaces).

```
// BAD: tight coupling
class OrderService { private SqlRepository _repo = new SqlRepository(); }
// GOOD: depend on interface (inject via constructor)
class OrderService { public OrderService(IRepository repo) { _repo = repo; } }
```

Section 5: Exception Handling

Q28: How do you create a Custom Exception?

Inherit from Exception. Always provide the standard constructors.

```
public class NotFoundException : Exception {
    public NotFoundException(string message) : base(message) { }
}
throw new NotFoundException("User not found");
```

- ⇨ *Custom exceptions are used in Clean Architecture to return meaningful domain errors (NotFoundException, ValidationException, etc.) instead of generic exceptions.*

Q29: What is the difference between throw and throw ex?

throw → rethrows the original exception and PRESERVES the original stack trace.
throw ex → resets the stack trace — you LOSE the original error location. Avoid this.

```
catch (Exception ex) { throw; }  // GOOD — preserves stack trace
```

Section 6: struct vs class vs record

Q30: What is the difference between struct and class?

struct — Value type. Stored on the Stack. Copied by value. No inheritance. Best for small, simple data (Point, Color, Money).

class — Reference type. Stored on the Heap. Copied by reference. Supports inheritance, polymorphism.

```
struct Point { public int X; public int Y; }
Point p1 = new Point { X=1, Y=2 };
Point p2 = p1; p2.X = 99; // p1.X is still 1 (copied by value)
```

⇨ **When to use struct:** small data, immutable values, performance-critical code (no GC overhead), no need for inheritance.

Q31: What is a record and how is it different from class? (C# 9+)

A record is a reference type (like class) but designed for immutable data with value-based equality.

Key differences from class:

- Equality is based on VALUES not reference (two records with same data are equal)
- Immutable by default (init-only properties)
- Built-in ToString(), GetHashCode(), Equals() auto-generated
- Supports with-expressions to create modified copies

```
public record Person(string Name, int Age);
var p1 = new Person("Samaa", 25);
var p2 = new Person("Samaa", 25);
Console.WriteLine(p1 == p2); // TRUE (value equality)
var p3 = p1 with { Age = 26 }; // new copy with changed Age
```

When to use record: DTOs, API responses, value objects, any data that should not change after creation.

Quick Comparison:

	class	struct	record
Type	Reference	Value	Reference
Equality	Reference	Value	Value (auto)
Mutable?	Yes	Yes (avoid)	No (immutable)

Inheritance	Yes	No	Yes (limited)
Use for	Business logic	Small data	DTOs / responses

Section 7: Type System & Casting

Q32: What is the difference between is and as operators?

Both are used for type checking and casting, but they behave differently.

- ⇨ **is** → checks if an object is of a certain type
- ⇨ Returns true/false --- does NOT throw an exception.
- ⇨ In modern C# (pattern matching), it can also cast in one step.

```
if (obj is Dog d) {
    d.Bark(); // safe — d is already cast
}
```

- ⇨ **as** → attempts to cast an object to a type.
- ⇨ Returns null if the cast fails — does NOT throw an exception.

```
Dog d = obj as Dog;
if (d != null) { d.Bark(); }
```

	Is	as
Purpose	Type check (+ optional cast)	Safe cast attempt
If cast fails	Returns false	Returns null
If cast success	True	Object
Throws exception?	No	No
Works with value types?	Yes (values & references)	No (reference types only)

Rule of thumb:
Prefer is with pattern matching (modern C#).
Use as when you need to check for null after casting.

Q33: What is the difference between == and .Equals()?

- ⇨ **== (operator)** → by default compares references for objects (are they the same object in memory?).
- ⇨ **.Equals()** → by default also compares references, BUT can be overridden to compare values.

	class (default)	Record	string
==	Reference equality	Value equality	Value equality

.Equals()	Reference equality	Value equality	Value equality

Rule of thumb:

- ⇨ *override .Equals() (and GetHashCode()) in a class when you want value-based comparison. Records do this automatically.*

Q34: What is Boxing and Unboxing?

Boxing → converting a value type (stored on the Stack) to a reference type (stored on the Heap). Happens implicitly.
Unboxing → converting from reference type to value type. Must be done explicitly with a cast.
Why does it matter?

- Boxing/Unboxing has a performance cost (heap allocation + GC pressure)
- Avoid it in performance-critical code — use generics instead (List<int> not ArrayList)

	Boxing	Unboxing
Direction	Value type → Reference type	Reference type → Value type
Implicit/Explicit	Implicit	Explicit (cast required)
Performance impact	Heap allocation	Type check + copy

Q35: What are the Mutable variables vs immutable variables in C#?

Type / Concept	IMMUTABLE ✅	MUTABLE ⚡
String handling	string — new object on every change	StringBuilder — edits in place
Value types	int, double, bool, char, decimal — copied by value	— (value types are immutable by nature)
struct	readonly struct — compiler blocks all field changes	mutable struct ⚠️ — anti-pattern, avoid it
class	Manually: readonly fields + no setters + sealed	class — mutable by default, shared by reference
record	record — immutable by default, use with to copy-modify	record class with { get; set; } — explicitly mutable
Collections	ImmutableList<T>, ImmutableDictionary<K,V> (System.Collections.Immutable)	List<T>, Dictionary<K,V>, Array — add / remove / update freely

Type / Concept	IMMUTABLE ✅	MUTABLE ⚡
Fields & constants	const — fixed at compile time readonly — fixed after constructor	Regular fields — writable anytime
Span<T> / Memory<T>	ReadOnlySpan<T> / ReadOnlyMemory<T> — read only	Span<T> / Memory<T> — mutable views over existing data

The One-Liner Rule

Immutable → Once created, the value never changes. Any 'modification' returns a brand-new object.

Mutable → The object itself changes in memory. The reference stays the same but the data inside shifts.

Q36: What is IDisposable and how does the using statement work?

IDisposable is an interface with one method: Dispose().
It is used to release unmanaged resources (database connections, file streams, HTTP clients) that the Garbage Collector cannot clean up automatically.

The using statement guarantees that Dispose() is called automatically when the block exits — even if an exception is thrown. It is syntactic sugar for a try/finally block.

```
using (var conn = new SqlConnection(connString))
{ conn.Open(); /* work */ }  // Dispose() called automatically here
```

Why prefer it over Destructor?

- Destructor runs at an unpredictable time (GC decides). Dispose() runs immediately when you need it.
- IDisposable + using is the standard .NET pattern for resource management.

⇨ *Any class that holds a SqlConnection, FileStream, HttpClient, or DbContext should implement IDisposable.*